Persistent and Shared Objects in Trellis/Owl

DEC-TR-440

Patrick O'Brien, Bruce Bullis, Craig Schaffert

Digital Equipment Corporation
77 Reed Road, HLO2-3/M08
Hudson, MA 01749
(617)568-5824

July 18, 1986

Accepted at the International Workshop
on Object-Oriented Database Systems
September 23 - September 26, 1986
Asilomar Conference Center, Pacific Grove, California

Abstract

This paper discusses the design of an object-oriented database extension to Trellis/Owl, a strongly-typed object-oriented programming language developed at Digital. We use the abstract data typing mechanisms of Trellis/Owl to provide a database collection type which enables programs to share objects in a distributed workstation environment. The database is an object repository which enables Trellis/Owl programs to coordinate the sharing of persistent objects. The object-oriented database is not intended for conventional database applications such as online transaction processing, but rather for applications which have complex data structuring requirements, and which access relatively "large" objects over "long" periods of time.

1 Introduction

Trellis/Owl[1] [Schaffert *et al* 85] is an object-based language that incorporates a type hierarchy with multiple inheritance and compile-time type checking. The combination of features in the language facilitates the design, implementation, and evolution of large computer programs. Object-based languages are particularly well suited for applications having complex data structuring needs, such as computer-aided design, office automation, programming environments, and artificial intelligence. For many of these applications, sharing information is an essential requirement, and thus database system support has become an important feature. Conventional database models, however, do not adequately support the special requirements of these applications. To address these needs we are investigating the use of an object-oriented database within the Trellis/Owl system.

The database's primary role is to enable multiple Trellis/Owl programs to share objects. Trellis/Owl is currently a single-user system that provides automatic storage management and presents a one-level store model to the programmer. Many programs, however, will want to share objects with other programs. Ideally, we would like to provide a single address space for all objects and all programs, thus extending the one-level store model. Unfortunately, providing a one-level store model for multiple users is very difficult to implement, especially in a multiple machine environment.

Instead, we are designing and implementing an object repository as a mechanism for allowing programs to share information. The object repository is a container for objects which will be shared among programs. The design of an object repository introduces several issues that must be addressed.

Programmers will want to access objects which reside in the repository. They will also want to structure the information in the repository by grouping certain objects within

[1] Trellis is a trademark of Digital Equipment Corporation.

collections. Simply maintaining references from local programs to objects in the repository would not be adequate. We have added some Trellis/Owl types that provide an object-oriented database model, thus enabling programmers to structure and access data in a repository.

From a single program's point of view, objects can be stored, retrieved, and replaced in the repository. The relationships that exist between objects in local storage are preserved in the repository. Objects in the repository will be self-contained; that is, they can only reference other objects in the repository, not ones in local storage. The structure of the database for a particular repository is used to give hints for retrieval and storage of objects; for efficiency reasons, when an object is accessed we may want to transfer only a subset of the objects that it references. We are looking at several mechanisms for describing what portion of the workspace or repository should be transferred when a particular object is accessed.

For multiple programs, sharing objects becomes an important issue. Coordinating concurrent access to shared objects implies that we look at transactions, locking, and version control. The repository provides a "sharing by copy" semantics. Programs are given copies of the objects from the repository and they coordinate sharing amongst themselves. Updates to a shared object are not reflected immediately in every program which has a copy of the object: a program will see the new version of the object when it asks for another copy. Versions play an important role in many of the application areas we are targeting. Therefore, we would like to provide low level mechanisms to support version control as well.

Our goal is not to provide all of the capabilities that are required of a conventional database management system; for example, a system which is designed for online transaction processing. We are concentrating on the features essential to the class of applications described earlier. The role of the object repository is primarily for storing and retrieving relatively large complex objects, providing for the sharing of objects among programs, and periph-

erally for occasional associative searches.

In the next section we give an overview of the Trellis/Owl language. In section 3 we discuss the motivation for our design of the object repository. Section 4 presents the programmer's view of database capability in Trellis/Owl. Section 5 discusses problems introduced by going to a two-level store model as well as issues related to sharing. We conclude in section 6 with a discussion of future work.

2 Overview of Trellis/Owl

Objects in Trellis/Owl are used to model application entities. Every object has a *type* and the object is known as an *instance* of that type. A type characterizes the behavior of its instances by describing the operations that can manipulate those objects. Programs are organized around type declarations which group together the operations and other definitions that describe the behavior of instances of the type.

Objects are manipulated with operations. An operation is applied to its first argument, called the *controlling object*, whose type determines the code to be invoked. There are two kinds of operations: *instance* operations and *class* operations. Instance operations apply to individual instances of a type. Class operations apply to a type as a whole, not to a particular instance of the type.

Objects can be thought of as having distinct parts, similar to record fields or entity attributes. For each such part, an object needs a pair of operations: one to get the part's value and one to modify the part's value. Trellis/Owl's *components* provide a mechanism for specifying and implementing these operations.

Components may be implemented in two different ways. One way is to declare the component to be a *field*. A field is an actual "slot" in the physical representation of the object

that will hold the value of the component. The get and modify operations are automatically generated to fetch from and store to the correct slot in the object. Alternatively, the behaviors of the get and the modify operations can be described in terms of other components or operations. The implementer of a type is free to choose which components to implement as fields and which as computed values; and the user of the type does not see any distinction between field components and computed components.

Types describe the behavior of objects. Some descriptions are more general than others. For example, a type `Person` might describe persons in general, including their name, address, and age. A type `Student` could describe students, including their name, address, age, school, and grades. Since students are persons, we say that `Student` is a *subtype* of `Person`. Conversely, we say that `Person` is a *supertype* of `Students`.

In Trellis/Owl, if a type `S` is a subtype of type `T`, then `S` objects must behave like `T` objects and may be used anywhere a `T` object may be used. Objects of type `S` can also have additional behavior defined by `S`. Moreover, a type can be a subtype of more than one type. If `S` is a subtype of both `T` and `U`, then `S` objects must behave like `T` objects and like `U` objects.

Frequently, definitions in a subtype would be identical to definitions in a supertype. After all, a subtype must provide the same behavior as its supertypes. Trellis/Owl provides an *inheritance* mechanism which allows a subtype to share definitions with its supertypes. By default, a subtype inherits the definitions in its supertypes. Only additions to the supertypes' behavior need be specified in the subtype. Inheritance of the supertype's definition, however, can be blocked by rewriting the definition in the subtype.

Sometimes several types are very similar. For instance, sets of integers and sets of strings have similar behavior. We can use Trellis/Owl's *type generator* facility to define a family of closely related types. For example a `Set` type generator can be used to define the types `Set[Integer]`, `Set[String]`, and `Set[Person]`. The individual members of the family

are known as *parameterized types* and can be used like any other type.

Frequently, objects, such as sets and lists, can be viewed as containing other objects. *Iterator* operations provide the ability to extract the contents of these container objects one item at a time. Iterators are an important abstraction tool because they allow us to access elements of a container without revealing the implementation of the container. Generally, an iterator can incrementally *yield* a series of objects. These objects are handled one at a time by a `for` expression. The body of the `for` loop executes once for each object in the series, thus the iterator and the for-loop body act like co-routines.

Sometimes an operation can not perform its main task. For example, suppose we had an operation that searches a list for a particular object, and the operation is called to search for an object that is not in the list. The **search** operation can signal an *exception* to let the calling operation know that the search did not complete normally because the object was not found. The calling operation has the responsibility for handling the exception, that is, deciding what should happen when the exception has been signaled.

More information on Trellis/Owl is available in the language reference manual [Schaffert *et al* 85], the tutorial [O'Brien], and in [Schaffert *et al* 86].

3 Motivation for our approach

The primary reason for adding database capability to Trellis/Owl is to enable programs to share objects. We will discuss why extending the one-level store model is inappropriate for our project. As a partial solution we have decided to design a database-like object repository. We will describe the particular needs of the applications we are targeting and justify why the object repository is better suited than conventional database models. Finally, we will discuss the issue of integrating types and database collections of objects into a coherent model.

The ideal base on which to build the appropriate semantics for sharing would be to have all programs share a single address space. In a distributed system, programmers would not be forced to worry about whether objects were local or remote. Such a uniform distributed object system is the best generalization of a single node system because there is no change in programmer-visible semantics and is therefore simpler for the programmer to deal with.

Many questions arise in designing a uniform distributed system. For example, how does a programmer control the placement of objects? When an operation is applied to a remote object, where does the computation actually occur? When should objects be copied or moved? Moreover, mechanisms are needed to permit atomic updating of multiple objects and to allow recovery from node crashes or communications failures.

The answers to these questions are not obvious. For example, simple transactions based on locking can provide for atomic update and recovery, but they tend to force a much more serial schedule than is logically necessary. Further mechanisms are required to get around this problem [Weihl & Liskov] [Schwarz & Spector]. We believe that significant long term research is needed before such uniform distributed systems will be practical for day to day use.

In the meantime, we still want to share objects among users in a distributed system. A centralized object repository allows us to do this. Object repositories are easier to implement than global address spaces. We can use well-understood database technology to store and retrieve objects from a server, and there is no question about where computation occurs. Coordinating sharing is simpler in a more centralized system, and we do not have to invent an efficient distributed garbage collection algorithm.

There is of course a cost. An object repository complicates the programmer's view by introducing a second level of storage. Some objects are temporary, others are persistent and shareable, and decisions must be made about when to update objects in the repository. Moreover, systems based on an object repository are less robust since the repository can

be a single point of failure.

Nevertheless, we believe this solution meets the requirements of several significant application areas. Database system support has become an essential part of many computer applications which have extended beyond the more traditional commercial applications into areas such as computer-aided design, office automation, programming environments, and artificial intelligence.

The specific needs of these areas are somewhat different from conventional database applications such as transaction processing. The model of computation for conventional systems can be described as accessing lots of "small" records which are held for "short" periods of time. The emphasis is on throughput; there may be lots of contention for the same records; the volume of transactions is large; and response time is critical. The applications we are targeting, such as programming environments and computer-aided design, have a different emphasis and somewhat different requirements. They tend to access "large" complex objects and hold on to them for "long" periods of time. The emphasis is on sharing of large objects; there is probably not a great deal of contention for the same objects; support for versions is desirable; objects have complex data structures which are not easily modeled using conventional database models; and quick retrieval of bulk data is required [Batory & Kim] [Woelk *et al*].

The requirements of the applications we are targeting are different but not incompatible with conventional database systems. There are tradeoffs, however, and we will be choosing mechanisms that support the model of computation we believe exists for these applications.

3.1 Data structuring needs

The data structuring requirements of these applications necessitate a different approach for satisfying their database needs. The data structuring capabilities of current database systems do not adequately support the complexities and variations that occur in these

applications. The object-oriented data model of Trellis/Owl, however, is well suited to handling the complex data structuring needs of our target application areas. Therefore, we want to extend the data modeling capabilities of Trellis/Owl to the object repository.

Perhaps one of the most fundamental problems of conventional database models is their lack of a notion of an abstract object [McLeod & King 81]. The correspondence between an object in the application environment and records in the database is indirect. In conventional databases, application objects are represented as record structures and are related indirectly through common attribute values. For example, a student and her advisor would typically be associated through an attribute which represents the advisor; the student record and the advisor record would each contain a copy of the attribute. In the object-oriented model objects represent themselves instead of using some identifier to stand for them; thus making it possible to directly reference an object from a related one. This *referential transparency* is just one of many advantages that the object-oriented data model provides. But it is very important since any change in an object's state is automatically seen by all other objects which refer to it – unlike relational systems, where a change in the key value of an entity is not propagated automatically to other tuples sharing that value. Moreover, normalization, which further inhibits the notion of an abstract object, becomes a moot point.

Thus, one of our primary goals in the design of the object repository is to have the repository and programming language support the same data types and structures. This helps remove the *impedance mismatch* [Zaniolo *et al*] between the programming language data structures and the database data structures. The impedance mismatch is exacerbated when object-oriented languages are interfaced to conventional databases, since objects correspond to real world entities and this one-to-one correspondence is not maintained with records or tuples in a database[2]. Therefore, we allow programs to store and retrieve any

[2]Note that the reverse situation is probably equally undesirable. Interfacing a conventional language such as Pascal or C to an object-oriented database will present an impedance problem as well in the opposite direction.

type of object in the repository without having to translate between different data structures.

3.2 Differentiating types and collections

The concepts of types and subtype hierarchies in object-oriented programming languages, and classes and is-a hierarchies in semantic modeling languages, are important and somewhat related features. Our extension to Trellis/Owl to provide database capability will have to integrate these related concepts in a consistent manner. We propose differentiating between the concepts and providing separate mechanisms to model the different abstractions.

A common complaint about conventional database systems is that they do not cleanly separate type definition from data declaration [Copeland & Maier]. For example, in relational systems there is no distinction between a type `Student` and a relation that contains `Student` records. A single declaration defines both the record structure (tuple) and the container (set of tuples). Thus, redundant specification is necessary to declare several relations to hold `Student` records.

Semantic modeling languages introduced *classification* [Mylopoulos *et al*], which is an abstraction mechanism used to group entities with similar properties into "classes." But classification does not differentiate between the notion that an object has a type and the notion that we want to group objects into collections. Classification embodies these two distinct aspects: sometimes referred to as *intension* and *extension* [Albanio *et al*]. The *intensional* aspect describes the behavior of objects in a class and is analogous to the type mechanism in object-based languages. The *extensional* aspect groups objects in a collection so they can be referred to as a single unit. It is more analogous to a set in object-based languages and not a type.

Using classification, one cannot have two collections of students without having two distinct

types, similar to conventional database systems. Trellis/Owl, however, cleanly separates type definition from type instantiation and we also want to separate type definition from the definition of a collection of objects stored in the database.

Many semantic modeling languages provide a subtyping mechanism, sometimes referred to as an *is-a* hierarchy [Mylopoulos *et al*] [McLeod & King 81]. The is-a hierarchy further confounds the situation. For example, suppose we have a class `Grad_Students` which is *is-a* related to a class `Students`. This relationship introduces two constraints. First, the elements of `Grad_Students` are always a subset of the elements of `Students`: this is the extensional constraint. Second, the behavior of elements of `Grad_Students` must be a specialization of the behavior of elements of `Students`: this is the intensional constraint.

We believe that the reasons for subsetting collections of objects (extensional aspect) can be quite different from the reasons for constructing subtypes (intensional aspect). The subtype notion in programming languages has proven to be extremely useful. Subtyping and inheritance provide added flexibility for describing relationships between types and thus improve the structure of code. The subtype notion is usually a static aspect of the language – an object cannot change its type once it has been created. Subtypes are created to structure code for various reasons including specialization, implementation, and combination [Halbert & O'Brien 86].

A collection of objects, on the other hand, might be divided into subsets for reasons quite different from subtyping criteria. For example, employees might be partitioned into subsets that reflect the department they currently belong to. Or we may restrict membership to certain collections based on some predicate involving the value of an attribute, for example the age of a person. Unlike instances of a type, membership in a collection may change over time.

For these reasons, we have decided to distinguish between types and collections of objects, as well as their respective hierarchies. Types and the type hierarchy in Trellis/Owl deal

with the intensional aspect, and the collection and sub-collection mechanism, which we will introduce in the next section, deal with the extensional aspect. This distinction increases the modeling capability of the language because it allows the use of the type and type hierarchy mechanism independently of the collection and subcollection mechanism.

4 Database types in Trellis/Owl

Our objective is to integrate database capability into the Trellis/Owl language using existing language features. This allows programs to access and manipulate databases in the same manner as all other objects. We would like to adhere to the object-oriented and strongly typed nature of Trellis/Owl. Moreover, we do not want to separate the programming language and the database manipulation language. Initially we will be concentrating on providing features that are essential to the application areas we are targeting. Therefore, some of the features of a conventional database management system, such as a sophisticated query facility or declarative integrity constraints, will not be considered.

Our design achieves a tight integration of database management concepts within the object-oriented style of Trellis/Owl. This contrasts with traditional language/database interfaces that rely upon a set of function calls or a separate data manipulation language which has little or no interaction with other language features. We use existing facilities for declaration of types and variables, along with existing control structures for iteration and exception handling. The database extension to Trellis/Owl is presented to the programmer as new types added to the library.

The most important type we introduce is the notion of a *database collection*, which groups objects to be stored in the database. Database collections are analogous to relations in the relational database model. The objects stored in a database collection are instances of Trellis/Owl's types. Database collections can have sub-collection relationships: one collection can be a sub-collection of another collection with the implication that if B is a

sub-collection of A then a member of B is also a member of A. We also define the notion of a *repository* which is a group of database collections.

4.1 Database Collections

We define a new type generator, called `DB_Collection`, which generates database collection parameterized types. One can think of a database collection as simply a set of Trellis/Owl objects which happen to reside in a repository.

The database collection generator is very similar to the set generator. The set generator enables one to declare sets of some other type, say integers. For example:

```
var my_set_of_integers: Set[Integer];
```

defines the variable `my_set_of_integers` as a set of integers. Similarly,

```
var my_students: DB_Collection[Student];
```

would define `my_students` as a database collection of students. The type specified with the generator, in this case `Student`, gives the structure of the elements in the database collection.

If `Grad_Student` was defined as a subtype of `Student`, we could also insert `Grad_Student` objects into the database collection `my_students`. In fact, we can insert objects of any subtype of `Student` into `my_students`.

A database collection will have several operations that are provided: "add an element," "remove an element," "select a particular element," and so on.

The database collection type is defined as

```
type_module DB_Collection [entity_type: Type]

    ! operations for DB_Collections go here
    ! we describe them below
```

```
end type_module;
```

The parameter **entity_type** determines the type of objects which will be stored in the collection. For example, if we have

```
var my_students: DB_Collection[Student];
```

then **entity_type** would refer to **Student**.

One of the operations defined on the type **DB_Collection** is the **create** operation, which is used to generate a new instance of a database collection. The interface for the **create** operation is

```
operation create (Mytype,
                  name: String,
                  database: Repository,
                  indices: set[Component_Name],
                  super_collections: set[DB_Collection])
    returns (Mytype)
    signals (duplicate_name);
```

Several arguments are passed to the **create** operation. The **name** of the collection can be used to identify the collection at a later time. Names must be unique for each repository. A collection has to exist in an object repository so the **create** operation takes the repository as an argument. The **indices** argument is a set of component names which are defined for the type **entity_type** and which should be regarded as indices for this collection[3]. The **super_collections** argument specifies that this database collection is a sub-collection of the super_collections named in the set. The entity_type of this collection will have to be the same type or a subtype of each super_collections' respective entity_type.

The **insert** operation is used to add objects to a collection. Its interface is

[3] We do not have a key constraint on DB_Collections. The use of keys will be greatly diminished in an object-oriented database since uniqueness does not depend on data values. Moreover, programmers can always add an operation to the **entity_type** to check for uniqueness of some component attributes.

```
operation insert (me, entity: entity_type)
    signals (already_there);
```

If the object already exists in the collection the **already_there** exception will be raised.

The **remove** operation is used to delete objects from a collection. Its interface is

```
operation remove (me, entity: entity_type)
    signals (not_found);
```

The object **entity** is removed from the database collection. Note that this does not delete the object – it simply removes it from the collection.

There are two operations that can be used to retrieve objects from a collection. The **elements** operation is an iterator that returns all of the entities in the collection. Its interface is

```
operation elements (me)
    yields (entity_type);
```

To iterate over the elements in the student collection mentioned earlier, we could write

```
for st: Student in elements (my_students) do
    .
    .
    .
end for;
```

We would also like to provide associative retrieval of objects within collections. For example, we would like to add a **select** iterator defined as follows

```
operation select (me, p: DB_Predicate)
    yields (entity_type);
```

The **select** iterator takes a predicate as an argument and will return the objects in the collection for which this predicate evaluates to true. In section 6.1 we will describe an

approach to providing a mechanism within Trellis/Owl to deal with predicate objects.

Programs will rely on transactions, which will be discussed in section 5.2, to coordinate the updating of objects in the repository. After retrieving an object, a program can modify the object and the object will be automatically updated in the repository when the transaction completes. There is no explicit update or replace operation.

4.2 Database entities

The structure of a collection's elements is defined by the type that is passed to the database collection type generator. The type is defined using ordinary Trellis/Owl specifications. Objects of any type can be stored in collections. An example of a type whose instances might be stored in a database collection would be

```
type_module Student

component me.Student_no: Integer get_only;

component me.name: String;

component me.address: String;

component me.advisor: Professor;

component me.courses: Set[Course];


    ! some operations defined on student for example:

operation enroll (me, c: Course)
    signals (already_enrolled);


end type_module;
```

`Student` is an example of a type that might have some of its instances stored in a database collection. The component mechanism in Trellis/Owl enables programmers to view objects as having a record-like structure, similar to conventional database notation.

4.3 Object repositories

We introduce another type, `Repository`, which corresponds roughly to a database. An object repository is a collection of database collections.

The `create` operation will define a new object repository. Its interface is

```
operation create (Mytype, name: String)
    returns (Repository);
```

When creating an object repository, the programmer specifies a name which will be used as a file specification to indicate where the repository should reside.

The `locate` operation will find an existing object repository. Its interface is

```
operation locate (Mytype, name: String)
    returns (Repository)
    signals (not_found);
```

Programs would use this operation to access an existing repository that another program created.

As an example, suppose we wanted to access an existing repository called the `university`, and iterate through the collections on the repository. We could do this in Trellis/Owl as follows

```
var university_db: Repository;
university_db := locate(Repository, "university");
for db: DB_Collection in db_collections (university_db) do
    .
    .
```

```
end for;
```

The db_collections operation is an iterator which returns all of the database collections defined in the repository. Its interface is

```
db_collections (me)
    yields (DB_Collection);
```

We might also provide an operation to return a particular database collection. For example

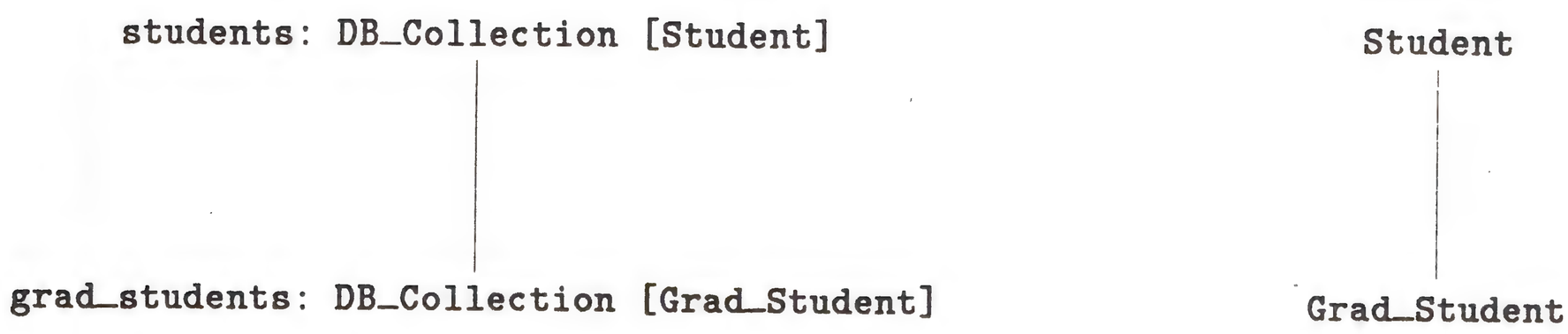
```
operation find_db_collection (me, name: String)
    returns (DB_Collection)
    signals (not_found);
```

The name of the collection is passed as an argument to the operation.

4.4 Sub-collection relationships

We will also enable programmers to define sub-collection relationships between database collections. Defining a collection b to be a sub-collection of collection a, will mean that the elements of b will be a subset of the elements of a. Therefore, if we iterate over the collection a, we will also get the elements of b returned. For this to work properly, there has to be a subtype relationship between the type of the elements of b and the type of the elements of a. In other words, b's elements have to be a subtype (or the same type) of a's elements.

Consider the following example:



The collection grad_students is a sub-collection of students, and the type Grad_Student is a subtype of Student. The collection students includes the elements of the collection Grad_Students. Note that the type Grad_Student must be a subtype (or the same type) of the type Student for the sub-collection relationship to be valid.

5 Transparency, sharing, and concurrency

5.1 An individual program and a repository

If we could have a single-level store, then using objects in a "database" would be exactly the same as using objects in local storage. Although we have settled on a two-level store, we would like to minimize the differences between using objects in a repository and using objects in local storage. We want to make the use of the repositories as transparent to the programmer as possible.

Trellis/Owl allows arbitrary sharing (referencing) of objects in its one-level store model. An object can be referenced by any number of objects including itself. If an object is modified, all objects that reference the object will see the change immediately. The arbitrary sharing semantics, however, can impact efficiency when we go to a two-level storage scheme.

For example, suppose we have a compound document object that references many other objects, for instance a heading, sections, etc. Those objects in turn reference other objects, for instance paragraphs, sentences, etc. If we restricted the sharing semantics of the

language so that no other object could reference any object referenced by the document, we could take advantage of this knowledge in the underlying storage scheme of the repository. We could cluster the objects, which constitute the document, together on disk.

But in a compound document system we might want to have different documents share a particular object, for example an appendix. But if arbitrary sharing is allowed, using the same clustering strategy will not work. We could try to cluster objects together as much as possible but some references to objects will get distributed across the disk.

There is a tradeoff between expressiveness and efficiency. But we do not want to restrict the sharing semantics of Trellis/Owl by not allowing arbitrary references. Therefore, we will preserve Trellis/Owl's sharing semantics. In the next section we will discuss another aspect of the sharing semantics: when do changes to shared objects get seen by different programs.

Another efficiency issue concerns what gets transferred when an object is referenced. A program accesses objects in a repository through that repository's database collections. The program will then be able to directly use the objects returned by a collection or any objects referenced by the returned objects. For efficiency reasons, the system must determine what actually gets transferred from the repository. Should it transfer a subset of what can be referenced or every object that can be referenced?

For example, suppose we have a database collection of books and we would like to modify a chapter in a particular book. If it is a large book with many chapters, it might take a long time to transfer all of the chapters of the book from the repository to local storage. In this example, we only want to change one chapter so much of this work is wasted effort.

For large complex objects, which reference other objects, we would like to transfer only the objects that are actually used. With smaller objects, it is more efficient to transfer a group of them, rather than transfer each object individually when it is referenced. Therefore, we would like to group objects into efficiently sized chunks such that most of the objects in

the chunk are actually used sometime. The system cannot predict the future object usage of a program, so we might like the programmer to give the system hints about a program's future use of objects.

The programmer could give the system *clustering hints* by describing which objects are frequently used together and therefore should be grouped together. An important question to be answered is whether we can statically decide which groups of objects should be transferred together or whether we will need a more dynamic, possibly adaptive, system. We will not be able to answer this question until we have had experience with implementing some applications.

One place to describe clustering hints may be in type definitions where the structure of objects is defined. Having one clustering for each type of object might make the underlying storage model easier. However, an object of a particular type may be used in many different ways and one clustering may not be optimal for all the different usages. Therefore, the type definition may not be the correct place to give these hints.

Another place to describe clustering hints is in the database collection definitions. When a collection is created the programmer already gives some usage hints by indicating the indices that can be used for associative retrieval of objects in the collection. The programmer could also give clustering hints for each collection. Therefore, a specific type would have more than one clustering strategy since instances of that type may reside in more than one collection. The objects in a particular collection may be used more uniformly than all instances of the type. Therefore, associating clustering hints with collections may give us more appropriate groupings. The underlying storage model would then have to handle more than one clustering for each type of object.

Our initial approach for clustering hints will be to enable the programmer to designate some objects as major (others being minor) with the intention of grouping minor objects with the major objects that reference them. Two major objects may share the same minor

object so there can be a possible conflict when deciding where a shared minor object should be placed.

To indicate which objects are major and which objects are minor, we are considering allowing the programmer to

- indicate within a type definition that all objects of this type are major objects
- indicate when creating a database collection that all instances of its element type should be considered major objects (whether they are in a collection or just referenced by an object in a collection)
- indicate when creating a database collection that all of its elements will be major objects

We will have to determine through experience the usefulness of each approach.

5.2 Multiple programs and a repository

One of the main reasons for having object repositories is to allow applications to share information. We must define what it means to share objects between applications and provide a synchronization mechanism for that sharing.

Object repositories will provide a *sharing by copy* semantics. When an application retrieves an object from a repository, the application gets a copy of the object in its local storage. The application can modify the object locally as it needs to. These changes are not reflected back to the repository or to other applications until the changed object is written back to the repository. After the object has been updated in the repository, other applications may get the new version of the object when they ask the repository for the object. Applications will have to coordinate their changes when sharing objects.

We recognize the need for two kinds of "transaction-like" behavior to provide object consistency. Low level, "instantaneous," serializable transactions are needed to synchronize actions on the data in the repository. For example, if two applications wanted to read and write the same object, the low level transactions will ensure that the two applications will not conflict with each other.

A number of applications also have "transaction-like" behavior. For example, a computer-aided design system may want to allow the design engineers to check out part of a design, work on the design, and then return the design. The semantics of this kind of "transaction" are application specific. Therefore, we want to provide a mechanism to allow applications to write these high level, application specific transactions.

Low level transactions can be implemented with locking, but we think that an optimistic approach will be more flexible. After considering some of the applications that we are trying to support, we believe that contention for objects will not be frequent, and, therefore, an optimistic approach will work well.

Initially, we are considering using timestamps and versions in our optimistic approach. If we assume that all objects are timestamped, the basic capabilities needed are

- read an object at a particular time
- write a set of objects if another set of objects is unchanged since a particular time

These operations are sufficient but would be awkward for applications to use because of the bookkeeping associated with keeping track of read and write sets. Therefore, we intend to design a *snapshot* mechanism to handle this bookkeeping for applications.

A snapshot of a repository is essentially a copy of the repository at a single instance of time. An application could create a snapshot, read objects from the snapshot, modify

objects, and write the snapshot back to the repository. All objects that are read from the snapshot are "as they were" when the snapshot was created.

The snapshot mechanism would keep track of which objects have been read and which objects have been changed or created. When the snapshot is written back to the repository, the mechanism will verify that none of the objects in the read set have been modified by another transaction since the snapshot was created. If any of these objects have been changed, the write-back will be undone.

Our snapshot mechanism will be designed to implement the low level, serializable transactions, but we think that applications will be able to use the mechanism to implement their own high level transactions. To provide some version support for the high level transactions, we will allow applications to create more than one snapshot at a time. We will also provide a way of comparing objects from different snapshots.

6 Future work

There are several areas which need further work. As mentioned in the previous section, we still have some design decisions to make for transactions, concurrency, and versions. We will experiment with different strategies for transferring objects between the repository and local storage. We would also like to design the predicates to be used for associative retrievals. Finally, we will be implementing a system to test our design and ideas. In this section we will discuss predicates and some implementation details.

6.1 Predicates

We would like to extract objects that satisfy a given condition from a database collection. One way to do this is to extract all the objects from the collection, and then discard those that do not match the condition. But since communication is expensive, we would rather

filter the collection at the repository, and only transfer the objects that are desired. This also enables the repository to use any indices that may exist to speed up locating the desired objects.

Evaluating a general programmer specified condition at the repository leads to some difficulties. For example, the condition might refer to non-shared objects that exist only on the client node. Fully solving this problem requires a uniform distributed object system, and we decided earlier that this was too ambitious a project for us.

Our solution is to restrict the allowed conditions to those that the repository can safely evaluate. (An alternative would be to automatically split the condition into two parts – a first pass filter that runs at the repository, and a second pass that runs on the client node.) There are two ways to limit the allowed conditions. We can test each condition presented to the repository and reject (at run time) those which it cannot evaluate, or we can insure (at compile time) that only proper conditions are presented to the repository. We have tentatively chosen the latter approach.

We will introduce a new type called `DB_Function`. An object of type `DB_Function` is intended to be applied to some object in a collection. Applying a DB function object will produce another object, but it has no side-effects. We will start off with a number of simple DB functions such as "make a constant", "extract this attribute", and so on. We will also provide a number of operations on `DB_Function` that construct more complex DB functions out of simple ones. We will probably create a subtype of `DB_Function` called `DB_Predicate` which will return boolean objects when applied to elements of a database collection. As an example of how predicates might be used, consider the following code which extracts math majors with high grade point averages from the collection `my_students`:

```
var p: DB_Predicate :=
            and (equal (attribute ("major"), constant ("math")),
                greater_than (attribute ("GPA"), constant (3.8));

for st: Student in select (my_students, p) do
```

```
        .
        .
        .
    end for;
```

We stated earlier that ad hoc retrieval of objects from collections will not be a high priority requirement for the applications we are targeting. The mechanism described above supports the kind of queries which most current programming language interfaces to databases provide. We believe that it will be adequate for our needs, especially since associative retrieval plays a much less central role in the model of computation for our expected applications.

6.2 Implementation

We have already implemented a prototype version of the object repository with a subset of the functionality. We wanted to experiment with the design of database collections to determine their suitability from a modeling standpoint. We did not experiment with transactions, versions, or snapshots in the prototype.

We built the prototype version on top of a relational database. Database collections were mapped onto relations and we achieved referential integrity with unique identifier fields for each object. If an object was not an element in a database collection, it was stored as a field in the record representing the object that referred to it. Additional relations were defined to model set relationships.

As an experimental application, we developed a database which stored source code for the Trellis programming environment. Trellis runs on workstations that are linked together in a local area network. The object repository resided either on a local workstation or on a remote mainframe. The records in the database tended to be fairly large by database standards: over 1K bytes in most cases. Most relations had several set relationships, for example to store comments or code.

We found that the performance of the object repository was too slow for our purposes. We have not yet tried to determine what caused the performance problems. We can make two observations, however. Shoehorning object data structures onto a relational database is less than ideal and a standard relational database has functionality that we do not really need. During the next few months we would like to gain a better understanding of where the real bottlenecks exist. We will use this data to help us design a system that is tailored for our repository's needs. Thus we expect to be looking closely at storage models and clustering strategies.

References

[Albanio *et al*] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems* **10**(2), June 1985.

[Batory & Kim] D.S. Batory and Won Kim. Modeling Concepts for VLSI CAD Objects *ACM Transactions on Database Systems* **10**(3), September 1985.

[Bever & Lockemann] Matrin Bever and Peter C. Lockemann. Database Hosting in Strongly-Typed Programming Languages. *ACM Transactions on Database Systems* **10**(1), March 1985.

[Borgida] Alexander Borgida. Features of languages for the development of information systems at the conceptual level. *IEEE Software* **2**(1), January 1985.

[Copeland & Maier] George Copeland and David Maier. Making Smalltalk a Database System. *SIGMOD '84 Proceedings*, Boston, MA. June, 1984.

[Halbert & O'Brien 86] D. Halbert and P. O'Brien. Using types and inheritance in object-oriented languages. Digital Equipment Corporation technical report DEC-TR-437, April 1986.

[McLeod & King 84] D. McLeod and R. King. Semantic Database Models. In *Principle of Database Design*, S.B. Yao, Ed., Prentice-Hall, 1984.

[McLeod & King 81] D. McLeod and R. King. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems* **6**(3), September 1981.

[Mylopoulos *et al*] John Mylopoulos, P. Bernstein, and H.K.T. Wong. A language facility for designing database-intensive applications. *ACM Transactions on Database Systems* **5**(2), June 1980.

[O'Brien] Patrick O'Brien. *Trellis Object-Based Environment: Language Tutorial.* Digital Equipment Corporation technical report DEC-TR-373, November 1985.

[Schaffert *et al* 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. *OOPSLA86 Proceedings*, Portland, Oregon, September 1986.

[Schaffert *et al* 85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. *Trellis Object-Based Environment: Language Manual.* Digital Equipment Corporation technical report DEC-TR-372, November 1985.

[Schrefl *et al*] M. Schrefl, A.M. Tjoa, and R.R. Wagner. Comparison-Criteria for Semantic Data Models. *International Conference on Data Engineering*, April 1984, 120-125.

[Schwarz & Spector] Peter M. Schwarz and Alfred Z. Spector. Synchronizing Shared Abstract Data Types. *ACM Transactions on Computer Systems* **2**(3), August, 1984.

[Weihl & Liskov] William Wiehl and Barbara Liskov. Implementation of Resiliant, Atomic Data Types. *ACM Transactions on Programming Languages and Systems* **7**(2), April, 1985.

[Woelk *et al*] Darrell Woelk, Won Kim, and Willis Luther. An Object-Oriented Approach to Multimedia Databases. MCC Technical Report Number: DB-172-85.

[Zaniolo *et al*] Carlo Zaniolo (Chair), Hassan Ait-Kaci, David Beech, Stephanie Cammarata, Larry Kerschberg, and David Maier. Object Oriented Database Systems and Knowledge Systems. *Expert Database Systems: Proceedings from the first international workshop*, October 1984, 49-65.